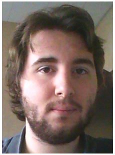
Sorting presorted data

Vincent Jugé

LIGM – Université Gustave Eiffel, ESIEE, ENPC & CNRS

14/06/2021

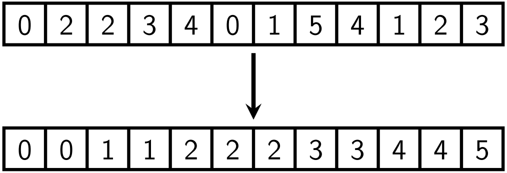
Joint work with N. Auger, C. Nicaud, C. Pivoteau & G. Khalighinejad



Université Gustave Eiffel

Sharif University
of Technology

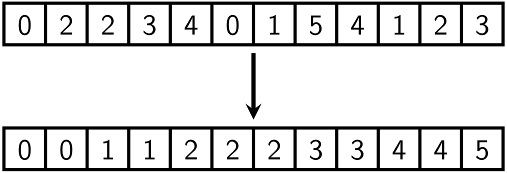
Sorting data



0	2	2	3	4	0	1	5	4	1	2	3

0	0	1	1	2	2	2	3	3	4	4	5

Sorting data



0	2	2	3	4	0	1	5	4	1	2	3

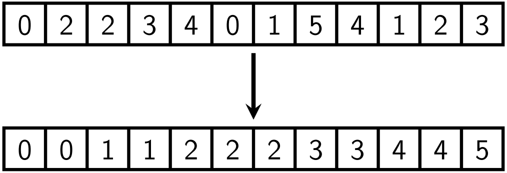
↓

0	0	1	1	2	2	2	3	3	4	4	5

MergeSort has a **worst-case time complexity** of $\mathcal{O}(n\log(n))$

Can we do better?

Sorting data



0 2 2 3 4 0 1 5 4 1 2 3

0 0 1 1 2 2 2 3 3 4 4 5

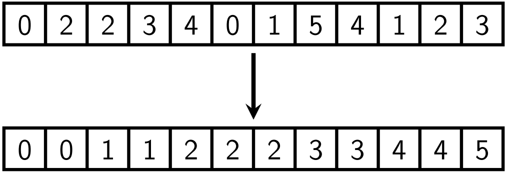
MergeSort has a **worst-case time complexity** of $\mathcal{O}(n \log(n))$

Can we do better? No!

Proof:

- There are $n!$ possible reorderings
- Each element comparison gives a 1-bit information
- Thus $\log_2(n!) \sim n \log_2(n)$ tests are required

Sorting data


0 2 2 3 4 0 1 5 4 1 2 3

0 0 1 1 2 2 2 3 3 4 4 5

MergeSort has a **worst-case time complexity** of $\mathcal{O}(n\log(n))$

Can we do better? No!

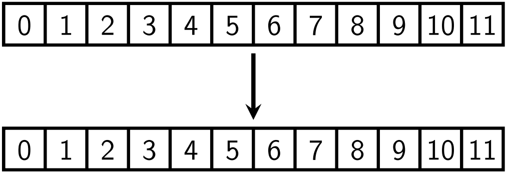
Proof:

- There are $n!$ possible reorderin
- Each element compar ation
- Thus $\log_2(n!) \sim n\log$ required

END OF TALK!

Cannot we ever do better?

In some cases, we should. . .

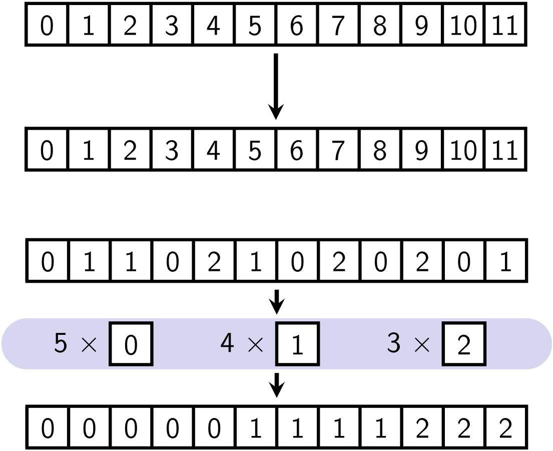


0 1 2 3 4 5 6 7 8 9 10 11

0 1 2 3 4 5 6 7 8 9 10 11

Cannot we ever do better?

In some cases, we should...



0	1	2	3	4	5	6	7	8	9	10	11

↓

0	1	2	3	4	5	6	7	8	9	10	11

0	1	1	0	2	1	0	2	0	2	0	1

↓

$5 \times \boxed{0}$ $4 \times \boxed{1}$ $3 \times \boxed{2}$

↓

0	0	0	0	0	1	1	1	1	2	2	2

Let us do better!

0	2	2	3	4	0	1	5	4	1	2	3

1. Chunk your data in **non-decreasing runs**

Let us do better!

4 runs of lengths **5**, **3**, **1** and **3**

0	2	2	3	4	0	1	5	4	1	2	3

1. Chunk your data in **non-decreasing runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)

Let us do better!

4 runs of lengths **5**, **3**, **1** and **3**

0	2	2	3	4	0	1	5	4	1	2	3

1. Chunk your data in **non-decreasing runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)

   Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i)$

   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

Let us do better!

4 runs of lengths 5, 3, 1 and 3

0	2	2	3	4	0	1	5	4	1	2	3

1. Chunk your data in **non-decreasing runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)

   Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i)$

   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

Theorem [1, 2, 4, 7, 11]

Some merge sort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

Let us do better!

4 runs of lengths **5**, **3**, **1** and **3**

0	2	2	3	4	0	1	5	4	1	2	3

1. Chunk your data in **non-decreasing runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)
   Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i)$
   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

Theorem [1, 2, 4, 7, 11]

TimSort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

Let us do better!

4 runs of lengths 5, 3, 1 and 3

0	2	2	3	4	0	1	5	4	1	2	3

1. Chunk your data in **non-decreasing runs**
2. New parameters: **Number of runs** (ρ) and their **lengths** ($r_1, \ldots, r_\rho$)
   Run-length entropy: $\mathcal{H} = \sum_{i=1}^{\rho} (r_i/n) \log_2(n/r_i)$
   $\leqslant \log_2(\rho) \leqslant \log_2(n)$

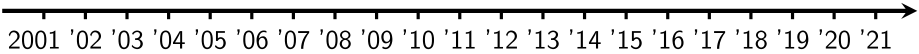
Theorem [1, 2, 4, 7, 11]

TimSort has a **worst-case time complexity** of $\mathcal{O}(n + n\,\mathcal{H})$

We cannot do better than $\Omega(n + n\,\mathcal{H})$![4]

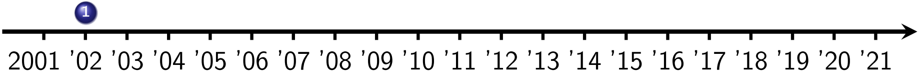
- Reading the whole input requires a time $\Omega(n)$
- There are X possible reorderings, with $X \geqslant 2^{1-\rho}\binom{n}{r_1 \ldots r_\rho} \geqslant 2^{n\mathcal{H}/2}$

A brief history of TimSort



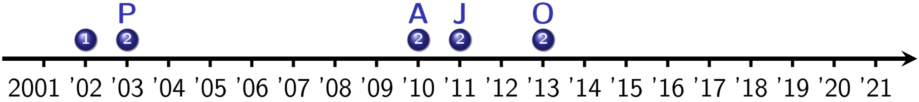
2001 '02 '03 '04 '05 '06 '07 '08 '09 '10 '11 '12 '13 '14 '15 '16 '17 '18 '19 '20 '21

A brief history of TimSort



❶ Invented by **Tim Peters**[3]

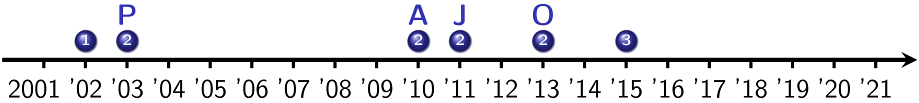
A brief history of TimSort



1. Invented by **Tim Peters**[3]
2. Standard algorithm in **Python**

   ———————— for non-primitive arrays in **Android**, **Java**, **Octave**

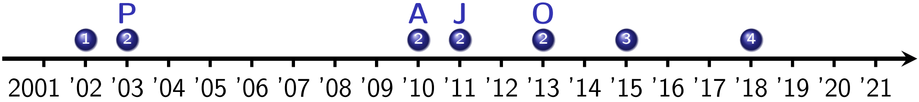
A brief history of TimSort



1. Invented by **Tim Peters**[3]
2. Standard algorithm in **Python**

   ——————— for non-primitive arrays in **Android**, **Java**, **Octave**
3. 1st worst-case complexity analysis[6] – TimSort works in time $\mathcal{O}(n \log n)$

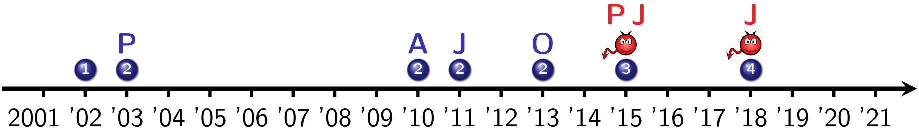
A brief history of TimSort



1. Invented by **Tim Peters**[3]
2. Standard algorithm in **Python**

   ——————— for non-primitive arrays in **Android**, **Java**, **Octave**
3. 1st worst-case complexity analysis[6] – TimSort works in time $\mathcal{O}(n \log n)$
4. Refined worst-case analysis[7] – TimSort works in time $\mathcal{O}(n + n\,\mathcal{H})$

A brief history of TimSort



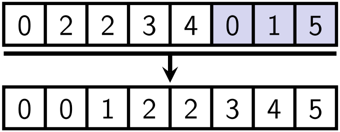
1. Invented by **Tim Peters**[3]
2. Standard algorithm in **Python**

   ———————— for non-primitive arrays in **Android**, **Java**, **Octave**
3. 1st worst-case complexity analysis[6] – TimSort works in time $\mathcal{O}(n \log n)$
4. Refined worst-case analysis[7] – TimSort works in time $\mathcal{O}(n + n\,\mathcal{H})$

😈 Bugs uncovered in Python & Java implementations[5, 7]

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs



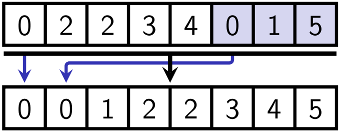
0	2	2	3	4	0	1	5

0	0	1	2	2	3	4	5

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

☛ **Stable** algorithm
(good for **composite** types)

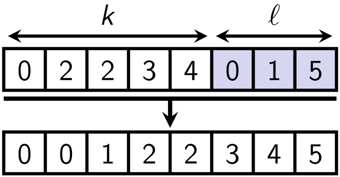

0 2 2 3 4 0 1 5

0 0 1 2 2 3 4 5

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

☛ **Stable** algorithm
(good for **composite** types)



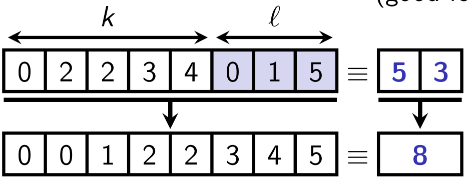
1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   $\Big\}$ **Merge cost:** $k+\ell$

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

☛ **Stable** algorithm
(good for **composite** types)



1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

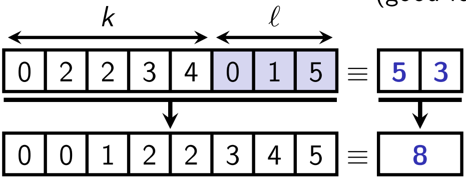
   $\Big\}$ **Merge cost:** $k+\ell$

2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only

The principles of TimSort and its variants (1/2)

Algorithm based on **merging** adjacent runs

☛ **Stable** algorithm
(good for **composite** types)



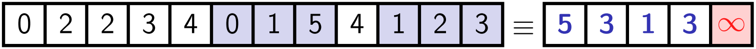
1. **Run merging** algorithm: standard + many optimizations
   - time $\mathcal{O}(k+\ell)$
   - memory $\mathcal{O}(\min(k,\ell))$

   $\Big\}$ **Merge cost:** $k+\ell$
2. **Policy** for choosing runs to merge:
   - depends on **run lengths** only
3. **Complexity analysis:**
   - ☛ Evaluate the **total merge cost**
   - ☛ Forget array values and only work with **run lengths**

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}

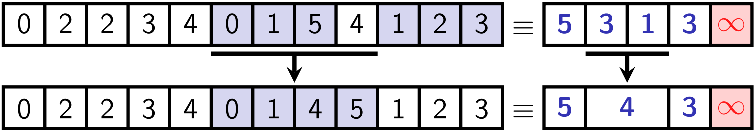


0	2	2	3	4	0	1	5	4	1	2	3	≡	**5**	**3**	**1**	**3**	∞

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



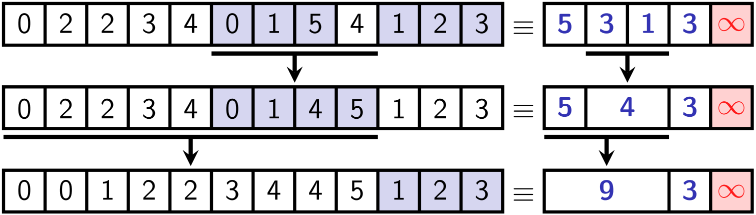
0 2 2 3 4 0 1 5 4 1 2 3 $\equiv$ 5 3 1 3 ∞

0 2 2 3 4 0 1 4 5 1 2 3 $\equiv$ 5 4 3 ∞

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1 + \sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



0 2 2 3 4 0 1 5 4 1 2 3 ≡ 5 3 1 3 ∞

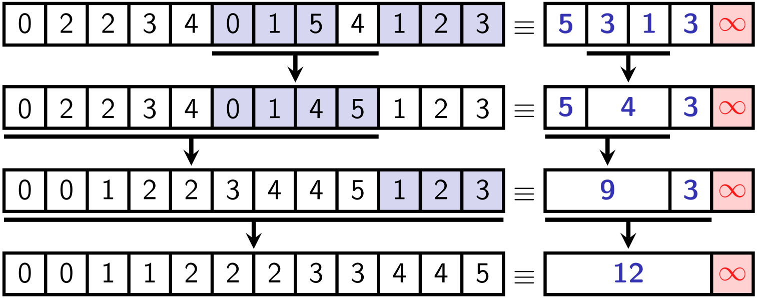
0 2 2 3 4 0 1 4 5 1 2 3 ≡ 5 4 3 ∞

0 0 1 2 2 3 4 4 5 1 2 3 ≡ 9 3 ∞

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



0 2 2 3 4 0 1 5 4 1 2 3 ≡ **5** **3** **1** **3** ∞

0 2 2 3 4 0 1 4 5 1 2 3 ≡ **5** **4** **3** ∞

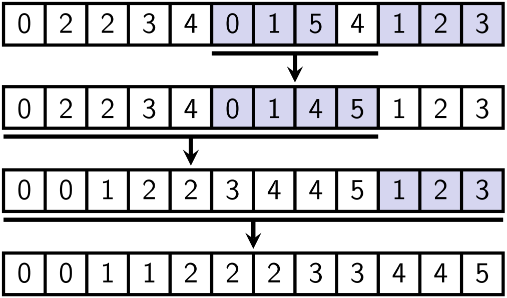
0 0 1 2 2 3 4 4 5 1 2 3 ≡ **9** **3** ∞

0 0 1 1 2 2 2 3 3 4 4 5 ≡ **12** ∞

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



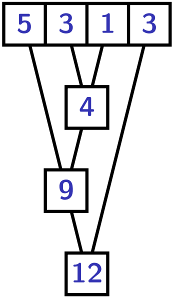
0 2 2 3 4 0 1 5 4 1 2 3

0 2 2 3 4 0 1 4 5 1 2 3

0 0 1 2 2 3 4 4 5 1 2 3

0 0 1 1 2 2 2 3 3 4 4 5

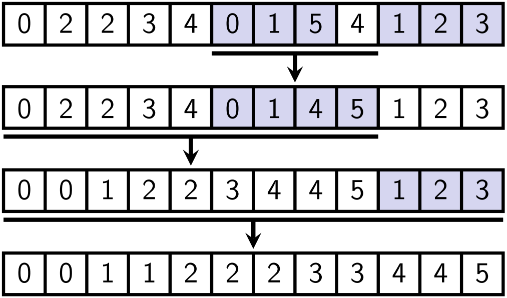
Merge tree



The principles of TimSort and its variants (2/2)

Run merge policy of α-**merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



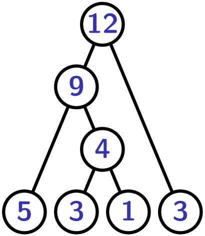
0 2 2 3 4 0 1 5 4 1 2 3

0 2 2 3 4 0 1 4 5 1 2 3

0 0 1 2 2 3 4 4 5 1 2 3

0 0 1 1 2 2 2 3 3 4 4 5

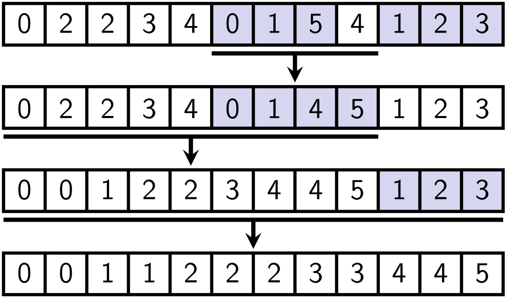
Merge tree



The principles of TimSort and its variants (2/2)

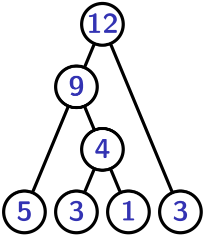
Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



0	2	2	3	4	0	1	5	4	1	2	3
0	2	2	3	4	0	1	4	5	1	2	3
0	0	1	2	2	3	4	4	5	1	2	3
0	0	1	1	2	2	2	3	3	4	4	5

Merge tree



× × × ×

$2 + 3 + 3 + 1$

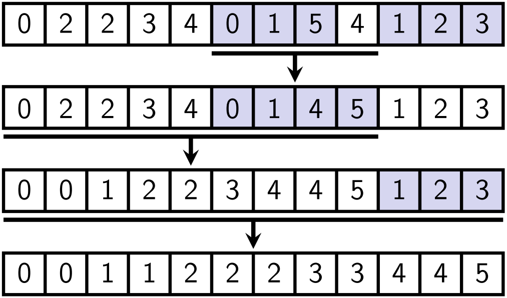
$=$

merge cost

The principles of TimSort and its variants (2/2)

Run merge policy of **α-merge sort**[9] for $\alpha = \phi = (1+\sqrt{5})/2 \approx 1.618$:

- Find the least index k such that $r_k \leqslant \alpha r_{k+1}$ or $r_k \leqslant r_{k+2}$
- Merge the runs R_k and R_{k+1}



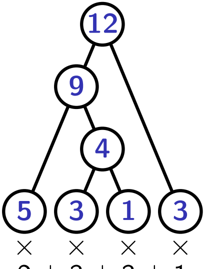
0 2 2 3 4 0 1 5 4 1 2 3

0 2 2 3 4 0 1 4 5 1 2 3

0 0 1 2 2 3 4 4 5 1 2 3

0 0 1 1 2 2 2 3 3 4 4 5

Merge tree



$2+3+3+1$
$=$
merge cost

$\alpha \geqslant \phi \Rightarrow k^{\text{new}} \geqslant k^{\text{old}} - 1$ after each merge

$\Rightarrow$ one can use **stack-based** implementations of α-merge sort

Fast growth in merge trees (1/2)

Theorem [11]

In merge trees induced by α-merge sort for $\alpha \geqslant \phi$, each node is at least $(\alpha+1)/\alpha$ times larger than its great-grandchildren

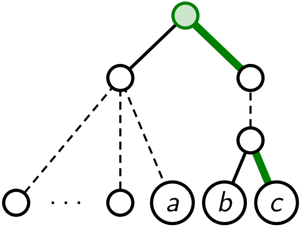
Fast growth in merge trees (1/2)

Theorem [11]

In merge trees induced by α-merge sort for $\alpha \geqslant \phi$, each node is at least $(\alpha + 1)/\alpha$ times larger than its great-grandchildren

Proof:

○ $\geqslant a + c \geqslant 2c$



Fast growth in merge trees (1/2)

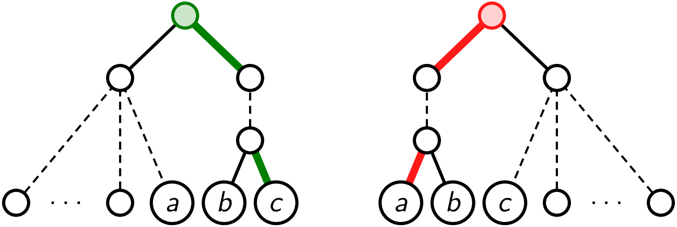
Theorem [11]

In merge trees induced by α-merge sort for $\alpha \geqslant \phi$, each node is at least $(\alpha+1)/\alpha$ times larger than its great-grandchildren

Proof:

🟢 $\geqslant a + c \geqslant 2c$

🔴 $\geqslant a + \max\{b, c\}$
$\geqslant (\alpha+1)a/\alpha$



Fast growth in merge trees (1/2)

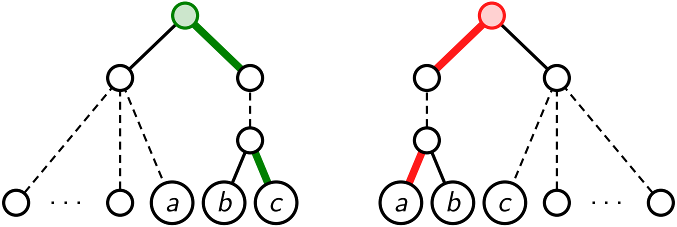
Theorem [11]

In merge trees induced by α-merge sort for $\alpha \geqslant \phi$, each node is at least $(\alpha+1)/\alpha$ times larger than its great-grandchildren

Proof:

🟢 $\geqslant a + c \geqslant 2c$

🔴 $\geqslant a + \max\{b, c\}$
$\geqslant (\alpha+1)a/\alpha$



Corollary:

- Each run R lies at depth $\mathcal{O}(1 + \log(n/r))$
- α-merge sort has a merge cost $\mathcal{O}(n + n\mathcal{H})$

Fast growth in merge trees (2/2)

Fast-growth property

A merge algorithm **A** has the **fast-growth property** if

- there exists an integer $k \geqslant 1$ and a real number $\varepsilon > 1$ such that
- in each merge tree induced by **A**,

going up k times multiplies the node size by ε or more

Fast growth in merge trees (2/2)

Fast-growth property

A merge algorithm **A** has the **fast-growth property** if

- there exists an integer $k \geqslant 1$ and a real number $\varepsilon > 1$ such that
- in each merge tree induced by **A**,

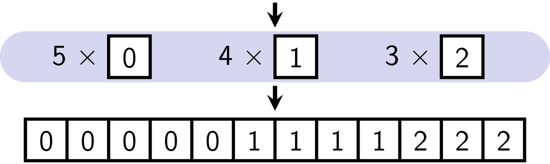
going up k times multiplies the node size by ε or more

Theorem (continued)

Timsort[3], **α-merge sort**[9] (when $\alpha \geqslant \phi$), **adaptive Shivers sort**[10], **Peeksort** and **Powersort**[8] have the fast growth-property

Corollary: These algorithms work in time $\mathcal{O}(n + n\mathcal{H})$

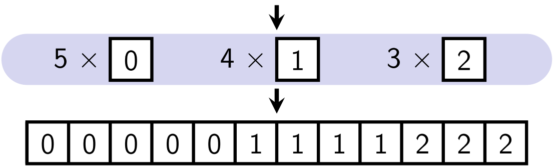
What about $\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{2}\boxed{1}\boxed{0}\boxed{2}\boxed{0}\boxed{2}\boxed{0}\boxed{1}$?



5 × $\boxed{0}$ 4 × $\boxed{1}$ 3 × $\boxed{2}$

$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{2}\boxed{2}\boxed{2}$

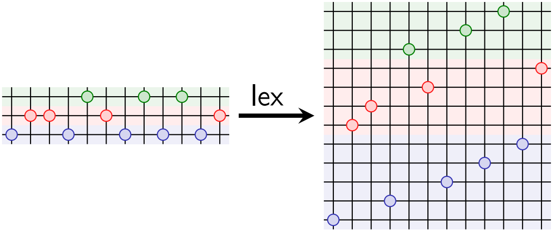
What about 0 1 1 0 2 1 0 2 0 2 0 1 ?



5 × 0 4 × 1 3 × 2

0 0 0 0 0 1 1 1 1 2 2 2

Few **runs** vs few **values**:



What about $\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{2}\boxed{1}\boxed{0}\boxed{2}\boxed{0}\boxed{2}\boxed{0}\boxed{1}$?

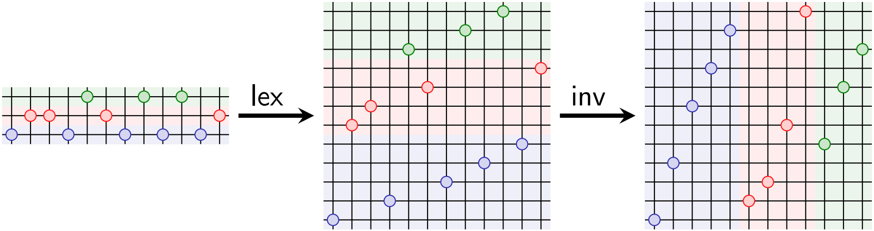
$$\downarrow$$

$5 \times \boxed{0}$ $4 \times \boxed{1}$ $3 \times \boxed{2}$

$$\downarrow$$

$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{2}\boxed{2}\boxed{2}$

Few **runs** vs few **values** vs few **dual runs**:



Let us do better, dually!

3 dual runs of lengths **5**, **4** and **3**

0	1	1	0	2	1	0	2	0	2	0	1

1. Chunk your data in non-decreasing, non-overlapping **dual runs**
2. New parameters: **Number of dual runs** ($\rho^\star$) and their **lengths** ($r_i^\star$)

   Dual-run entropy: $\mathcal{H}^\star = \sum_{i=1}^{\rho^\star} (r_i^\star / n) \log_2(n / r_i^\star)$
   $\leqslant \log_2(\rho^\star) \leqslant \log_2(n)$

Let us do better, dually!

3 dual runs of lengths **5**, **4** and **3**

0	1	1	0	2	1	0	2	0	2	0	1

1. Chunk your data in non-decreasing, non-overlapping **dual runs**
2. New parameters: **Number of dual runs** ($\rho^\star$) and their **lengths** ($r_i^\star$)

   Dual-run entropy: $\mathcal{H}^\star = \sum_{i=1}^{\rho^\star} (r_i^\star / n) \log_2(n / r_i^\star)$
   $\leqslant \log_2(\rho^\star) \leqslant \log_2(n)$

Theorem [11]

Every **fast-growth** merge sort requires $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons if it uses **Timsort's optimized run-merging routine**

and we still cannot do better than $\Omega(n + n\,\mathcal{H}^\star)$

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\,\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\mathcal{H}^\star)$ comparisons
- Both its **merging policy** and **merging routine** are great!

Conclusion

- **TimSort** is good in practice **and** in theory: $\mathcal{O}(n + n\,\mathcal{H})$ merge cost
  $\mathcal{O}(n + n\,\mathcal{H}^\star)$ comparisons
- Both its **merging policy** and **merging routine** are great!

Some references:

[1] *Optimal computer search trees and variable-length alphabetical codes*, Hu & Tucker (1971)

[2] *A new algorithm for minimum cost binary trees*, Garsia & Wachs (1973)

[3] Tim Peters' description of TimSort, `svn.python.org/projects/python/trunk/Objects/listsort.txt` (2001)

[4] *On compressing permutations and adaptive sorting*, Barbay & Navarro (2013)

[5] *OpenJDK's java.utils.Collection.sort() is broken*, de Gouw et al. (2015)

[6] *Merge strategies: from merge sort to TimSort*, Auger et al. (2015)

[7] *On the worst-case complexity of TimSort*, Auger et al. (2018)

[8] *Nearly-optimal mergesorts*, Munro & Wild (2018)

[9] *Strategies for stable merge sorting*, Buss & Knop (2019)

[10] *Adaptive ShiversSort: an alternative sorting algorithm*, Jugé (2020)

[11] *Galloping in natural merge sorts*, Jugé & Khalighinejad (2021^+)

MERCI POUR VOTRE ATTENTION !
NE POSEZ PAS DE QUESTIONS DIFFICILES S'IL VOUS PLAÎT !